Static Analysis of List-Manipulating Programs via Bit-Vectors and Numerical Abstractions [*]

Liqian Chen[†‡] lqchen@nudt.edu.cn
Renjian Li[†] li.renjian@gmail.com
Xueguang Wu[†] xueguangwu@sina.cn
Ji Wang[†‡] wj@nudt.edu.cn

[†] National University of Defense Technology, Changsha 410073, China
[‡] National Laboratory for Parallel and Distributed Processing, Changsha 410073, China

ABSTRACT

We present an approach under the framework of abstract interpretation to analyze list-manipulating programs by combining shape and numerical abstractions. The analysis automatically divides a list into non-overlapping list segments according to the reachability property of pointer variables to list nodes. The list nodes in each segment are abstracted by a bit-vector wherein each bit corresponds to a pointer variable and indicates whether the nodes can be reached by that pointer variable. Moreover, for each bit-vector, we introduce an auxiliary integer variable, namely a counter variable, to record the number of nodes in the segment abstracted by that bit-vector. On this basis, we leverage the power of numerical abstractions to discover numerical relations among counter variables, so as to infer relational length properties among list segments. Our approach stands out in its ability to find intricate properties that involve both shape and numerical information, which are important for checking program properties such as memory safety and termination. A prototype is implemented and preliminary experimental results are encouraging.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms

Languages, Reliability, Theory, Verification

Keywords

Static analysis, Abstract interpretation, Lists, Abstract domains, Shape analysis

[*]This work is supported by the National Natural Science Foundation of China (NSFC) under Grant Nos. 61120106006, 61202120, 91118007, and the National High Technology Research and Development Program of China (863 Program) under Grant No. 2011AA010106.

1. INTRODUCTION

Invariants involving both shape and numerical information are crucial for checking nontrivial program properties in heap manipulating programs, such as memory safety, termination, bounded size of heap memory. However, automatically inferring such invariants is challenging, especially for programs manipulating dynamic linked data structures. In this paper, we consider the problem of analyzing programs manipulating lists. And inferring such invariants over lists requires considering both the shape of lists and the numerical information over the number of the list nodes.

Shape analysis provides a powerful approach to generate shape invariants, and much progress has been achieved in shape analysis in the past two decades [1][23]. However, shape analysis itself has limited ability in inferring non-trivial properties involving numerical information such as "*the length of the segment between p and q is n where n is an integer greater than 1*". On the other hand, numerical static analysis by abstract interpretation [7] is widely adopted to automatically generate numerical invariants for programs. However, most existing abstract domains focus on purely numerical properties and thus are specific for analyzing numerical programs. A recent interesting trend is to combine these two techniques, using shape analysis to generate shape invariants and using numerical abstract domains to track numerical relationships [3][5][10][22]. The key technical issue here is how to interact effectively between the shape aspect and the numerical aspect. Although several generic frameworks for the combination have been proposed [3][10], tighter bidirectional coupling between the two aspects still needs further considerations for the selected shape abstraction and numerical abstraction. For instance, shape abstraction needs to be enhanced to support numerical aspects while numerical abstraction also needs to be adapted with respect to the semantics of shape abstraction. And transfer functions for the combined domain should be designed by taking into account both the shape and numerical information at the same time.

In this paper, we present an approach in the framework of abstract interpretation to combine shape and numerical abstractions for analyzing programs manipulating lists. First, for the shape of a list, we propose a lightweight shape abstraction based on bit-vectors, upon the insight that list nodes in a list can be naturally grouped into nonoverlapping list segments according to the reachability property of pointer variables to list nodes. Each list segment which includes those list nodes that can be reached by the same set of pointer variables, is abstracted by one bit-vector wherein each bit corresponds to a pointer variable in the program. From the numerical aspect, in order to track the number of list nodes in each list segment, we introduce an auxiliary (nonnegative) integer counter variable for each bit-vector. And we apply numerical abstract domains to infer numerical relations among counter variables. Then, transfer functions for the combined domain are constructed in terms of transfer functions of the numerical domain upon the semantics of the shape abstraction. Specifically, in this paper we instantiate our approach by using a combination of intervals [6] and affine equalities [13] to conduct numerical abstractions. On this basis, a prototype is implemented and preliminary experimental results are presented on benchmark programs.

The rest of the paper is organized as follows. Section 2 describes a simple list-manipulating programming language. Section 3 presents a shape abstraction approach for lists based on bit-vectors. Section 4 presents a combined domain of intervals and affine equalities to conduct numerical abstractions over counter variables. Section 5 presents our prototype implementation together with preliminary experimental results. Section 6 discusses some related work before Section 7 concludes.

2. LIST-MANIPULATING PROGRAMMING LANGUAGE

We first present a small language that manipulates lists. The syntax of our language is depicted in Fig. 1. It is a simple procedureless sequential language with dynamic allocation and deallocation but no recursion. There is only one type of variables, i.e., pointer variables of LIST type, denoted as *PVar*. For the sake of simplicity, we first focus on non-circular singly-linked list.

The structure for list nodes contains a *next* field pointing to the successive list node, while all other fields are considered as data fields. The data fields are ignored in this paper, since we assume that operations over data fields have no influence over the shape of lists. We assume that there is at most one *next* operator in a statement and a pointer variable appears at most once in an assignment statement. All other cases could be transformed into this form by introducing temporary variables.

$$
\begin{array}{rcl}
p, q \in PVar & & \\
AsgnStmnt & := & p := null \mid p := q \mid p := q \rightarrow next \mid \\
 & & p \rightarrow next := null \mid p \rightarrow next := q \mid \\
 & & p := \text{malloc}() \mid \text{free}(p) \\
Cond & := & p == q \mid p == null \mid \neg Cond \mid \\
 & & Cond_1 \vee Cond_2 \mid Cond_1 \wedge Cond_2 \mid \\
 & & \text{true} \mid \text{false} \mid \text{brandom} \\
BranchStmnt & := & \textbf{if } Cond \textbf{ then } \{Stmnt;\}^* \ [\textbf{else } \{Stmnt;\}^* \ ] \textbf{ fi} \\
WhileStmnt & := & \textbf{while } Cond \textbf{ do } \{Stmnt;\}^* \textbf{ od} \\
Stmnt & := & AsgnStmnt \mid BranchStmnt \mid WhileStmnt \\
Program & := & \{Stmnt;\}^*
\end{array}
$$

Figure 1: Syntax of a list-manipulating program

3. LIST ABSTRACTION BASED ON BIT-VECTORS

First, we recall the definition of classic shape graph that is a graph used to represent the allocated memory in heap.

Definition 1. A shape graph for lists is a tuple $SG = \langle N, V, E \rangle$, where:

- N denotes the set of pointer variables and list nodes, and we utilize N_{nil} to denote $N \cup \{NULL\}$,
- $V \subseteq N$ denotes the set of pointer variables in the program,
- $E \subseteq N \times (N_{nil} - V)$ denotes the set of edges, which describes the points-to relations of pointer variables as well as successive relations between list nodes through the "*next*" field.

From the above definition, we can see that when using a standard shape graph to describe lists, we have to name explicitly all list nodes and store all the successive relations between nodes. Hence, using shape graph may cause heavy memory costs. To this end, we propose a lightweight approach to encode the shape information contained in a shape graph. First, we introduce a binary predicate $Reach(n, n')$ to describe the reachability property between two nodes $n, n' \in N$:

$$
\begin{aligned}
Reach(n, n') \triangleq \ & \exists k \in \mathbb{N}. \forall 0 \le i \le k. n_i \in N \wedge n_0 = n \wedge n_k = n' \wedge \\
& \forall 0 \leqslant j < k. \langle n_j, n_{j+1} \rangle \in E
\end{aligned}
$$

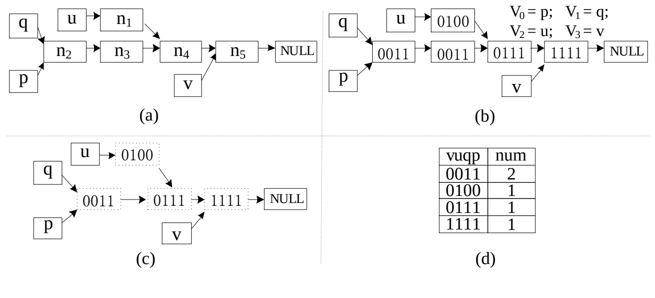
Obviously, $Reach(n, n') = true$ holds if and only if there exists a path from n to n' in the shape graph. We maintain a variable order for all pointer variables V in the program and use V_i to denote the i-th variable in V where $0 \le i \le |V| - 1$.

Definition 2. For each node $n \in (N - V)$, we define a so-called *Variable Reachability Vector (VRV)* $\mathbf{vec}_n \in \{0, 1\}^{|V|}$ that is a bit-vector of length $|V|$, where

$$\mathbf{vec}_n[i] = 1 \quad \textit{iff} \quad Reach(V_i, n) = true$$

We say V_i *reaches* list node n (or VRV $\mathbf{vec}_n$) if $Reach(V_i, n) = true$. We use bit-vector $\mathbf{0}$ as the VRV for those list nodes that can not be reached by any pointer variables. Let Γ denote the set of VRVs for all list nodes $n \in (N - V)$. For every $\mathbf{vec} \in \Gamma$, let $\mathcal{I}_{\mathbf{vec}}$ denote the set of the 1-bits in $\mathbf{vec}$: $\mathcal{I}_{\mathbf{vec}} \triangleq \{i \in \mathbb{N} \mid \mathbf{vec}[i] = 1\}$. In other words, $\mathcal{I}_{\mathbf{vec}}$ describes the set of the indices of those pointer variables that can reach $\mathbf{vec}$. If $i \in \mathcal{I}$, it means that V_i can reach $\mathbf{vec}$ (and the corresponding nodes). We use $\Gamma_i \triangleq \{\mathbf{vec} \mid \mathbf{vec}[i] = 1\}$ to denote the set of VRVs that the variable V_i can reach. In fact, Γ describes the reachability properties of all pointer variables to list nodes. Each VRV $\mathbf{vec}_n$ can be considered as an abstract node that represents the set of nodes which can be reached by the same set of pointer variables as node n.

Example 1. For the shape graph shown in Fig. 2 (a), suppose the variable ordering is $p \prec q \prec u \prec v$. Then the VRVs for this shape graph are shown in Fig. 2 (b). And we have $\Gamma = \{0011, 0100, 0111, 1111\}$; $\mathcal{I}_{0011} = \{0, 1\}, \mathcal{I}_{0100} = \{2\}, \mathcal{I}_{0111} = \{0, 1, 2\}, \mathcal{I}_{1111} = \{0, 1, 2, 3\}$; $\Gamma_0 = \{0011, 0111, 1111\}$, $\Gamma_1 = \{0011, 0111, 1111\}$, $\Gamma_2 = \{0111, 1111\}$, $\Gamma_3 = \{1111\}$.



vuqp	num
0011	2
0100	1
0111	1
1111	1

Figure 2: Example of variable reachability vectors for lists

Definition 3. Given two non-zero VRVs $\mathbf{vec}_1, \mathbf{vec}_2$,

- if $\mathcal{I}_{\mathbf{vec}_1} \subseteq \mathcal{I}_{\mathbf{vec}_2}$, we say $\mathbf{vec}_1$ can reach $\mathbf{vec}_2$, denoted as $\mathbf{vec}_1 \subseteq \mathbf{vec}_2$,
- if $\mathcal{I}_{\mathbf{vec}_1} \subset \mathcal{I}_{\mathbf{vec}_2}$, we say $\mathbf{vec}_1$ can strictly reach $\mathbf{vec}_2$, denoted as $\mathbf{vec}_1 \subset \mathbf{vec}_2$,

- if $\mathcal{I}_{\mathbf{vec}_1} \cap \mathcal{I}_{\mathbf{vec}_2} = \emptyset$, we say $\mathbf{vec}_1$ and $\mathbf{vec}_2$ can not reach each other, denoted as $\mathbf{vec}_1 \cap \mathbf{vec}_2 = \emptyset$.

For the example shown in Fig. 2, we have: $\mathbf{vec}_{0011} \subset \mathbf{vec}_{0111}$; $\mathbf{vec}_{0100} \subset \mathbf{vec}_{0111}$; $\mathbf{vec}_{0100} \cap \mathbf{vec}_{0011} = \emptyset$.

THEOREM 1. *Given two list nodes n_1, n_2 such that $\mathbf{vec}_{n_1} \neq \mathbf{vec}_{n_2}$ and $\mathbf{vec}_{n_1} \neq 0$, there exists one path from n_1 to n_2 if and only if $\mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2}$ holds.*

The bitwise set relations among VRVs implicitly characterize the reachability relations among nodes. All VRVs in Γ_i form a total order over $\subseteq$. Let $\mathbf{vec}_i^0$ denote the minimum element in Γ_i, then $\mathbf{vec}_i^0$ represents the VRV of the list node that variable V_i directly points to. For the example in Fig. 2, we have $\Gamma_0 = \{0011, 0111, 1111\}$, from which we can see that p directly points to 0011, since 0011 is the minimum element in Γ_0. Furthermore, from $\Gamma = \{0011, 0100, 0111, 1111\}$, we can see that

- p, q are alias, since in each VRV from Γ the bit corresponding to p is 1 if and only if the bit corresponding to q is 1;
- p cannot reach the node that is directly pointed to by u, since the bit corresponding to p in the minimum element of Γ_2 (i.e., 0100) is 0.

Definition 4. A set of VRVs Γ is *consistent*, if for arbitrary two distinct VRVs $\mathbf{vec}_{n_1}, \mathbf{vec}_{n_2} \in \Gamma$, $\mathbf{vec}_{n_1} \cap \mathbf{vec}_{n_2} = \emptyset \vee \mathbf{vec}_{n_1} \subset \mathbf{vec}_{n_2} \vee \mathbf{vec}_{n_2} \subset \mathbf{vec}_{n_1}$ holds.

THEOREM 2. *The set of VRVs of a singly-linked list is consistent.*

THEOREM 3. *A consistent set of VRVs Γ satisfies $|\Gamma| \leq 2|V|$.*

Definition 5. A set of VRVs with Counters (VRVCs) $\Gamma^+ \subseteq \Gamma \times \mathbb{N}$ is defined as a set of 2-tuples $\langle \mathbf{vec}, num \rangle$ where $\mathbf{vec} \in \Gamma$, $num \in \mathbb{N}$ standing for the number of the list nodes whose VRV is $\mathbf{vec}$.

The (consistent) set of VRVs with counters provides an exact abstraction for the shape of lists when ignoring the data contents. The list nodes are abstracted via VRVs (i.e., the **vec** component), the edges (i.e., the successive relations between nodes) are abstracted via the implicit bitwise subset relations of VRVs, and the number of the nodes that are reachable by the same set of pointer variables are described by the counters (i.e., the *num* component).

4. NUMERICAL ABSTRACTION OVER COUNTERS

For each $\mathbf{vec} \in \Gamma$, we introduce an auxiliary counter variable $t^{\mathbf{vec}} \in \mathbb{N}$ to denote the value of the corresponding *num* component (i.e., the number of the list nodes whose VRV is **vec**) of VRVCs. We maintain a bijection between **vec** and $t^{\mathbf{vec}}$. For each $t^{\mathbf{vec}}$, we use $VEC(t^{\mathbf{vec}})$ to obtain its corresponding bit vector **vec**. Furthermore, we introduce a special auxiliary variable $t^{0\ldots00} \in \mathbb{N}$ to specify memory leak (i.e., $t^{0\ldots00} > 0$). We use a lexicographic ordering on counter variables: $t^{0\ldots00} \prec t^{0\ldots01} \prec t^{0\ldots10} \prec \cdots \prec t^{1\ldots11}$. And $\{\langle \mathbf{vec}, t^{\mathbf{vec}} \rangle \mid t^{\mathbf{vec}} > 0\}$ represents the shape of a list, if it is consistent.

Since counter variables $t^{\mathbf{vec}} \in \mathbb{N}$ are numerical variables, we could leverage numerical abstraction techniques over $t^{\mathbf{vec}}$. In this paper, we present an abstract domain, namely the $\mathcal{CD}$ domain, to perform numeric abstraction over counter variables, which combines the interval abstract domain [6] and the affine equality abstract domain [13]. If the program has k pointer variables, we need introduce 2^k auxiliary counter variables. We choose intervals and affine equalities to construct the $\mathcal{CD}$ domain, because they are cheap in both time and memory, and bounds as well as equality relations are important for list-manipulating programs. However, it is also worthy noting that according to Theorem 3, most auxiliary counter variables equal to 0 and only linear number of counter variables with respect to $|V|$ need to be tracked.

4.1 Representation

We use intervals to track the range information of each counter variable $t^{\mathbf{vec}} \in \mathbb{N}$, and use affine equalities to track the relational information among those counter variables. Hence, each domain element P in the $\mathcal{CD}$ domain is described as an affine system $Ax = b$ in reduced row echelon form together with bounds for counter variables $x \in [c, d]$, where $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, c \in \mathbb{N}^n, d \in \{\mathbb{N}, +\infty\}^n, 0 \leq c \leq d$. It represents the set $\gamma(P) = \{x \in \mathbb{N}^n \mid Ax = b, c \leq x \leq d\}$ where each point x is a possible environment (or state), i.e., an assignment of nonnegative integer values to counter variables. For the sake of convenience, we use $EQS(P)$ to denote the affine equality part, and $ITV(P)$ to denote the interval part from the domain representation.

Given each $x \in \gamma(P)$, we can derive a set of VRVs from x: $VRV(x) = \{VEC(x_j) \mid x_j \geq 1\}$, as well as a set of VRVs with counters $VRVC(x) = \{\langle VEC(x_j), x_j \rangle \mid x_j \geq 1\}$. Obviously, if $VRV(x)$ is consistent, denoted as $wf(VRV(x))$, then $VRVC(x)$ describes a shape of singly-linked lists. And we use $\bar{\gamma}(P)$ to denote the obtained consistent set of VRVCs:

$$\bar{\gamma}(P) = \{VRVC(x) \mid x \in \gamma(P) \wedge wf(VRV(x))\}$$

$\bar{\gamma}(P)$ can be constructed from $\gamma(P)$ by considering the consistency among VRVs.

Example 2. Consider the program fragment

```
traverse($q$) {
  ① $p := q \rightarrow next$;
  ② while ($p \neq null$) {
       ③ $p := p \rightarrow next$; }
}
```

Assume q points to a list with length 9 before calling *traverse*(q) and the variable ordering is $p \prec q$. Then in the first iteration, at program point ③, we obtain the $\mathcal{CD}$ domain element $P = \{t^{10} = 1, t^{11} = 8; t^{10} \in [1, 1], t^{11} \in [8, 8]\}$. Then we know that $\bar{\gamma}(P)$ contains only one possible list shape: $\bar{\gamma}(P) = \{\{\langle 10, 1 \rangle, \langle 11, 8 \rangle\}\}$.

4.2 Domain Operations over Counters

In the $\mathcal{CD}$ domain, most domain operations (such as meet, join, inclusion, etc.) over counter variables can be directly constructed by combining the corresponding domain operations of the interval abstract domain and that of the affine equality abstract domain. E.g., the join operation in the $\mathcal{CD}$ domain consists of the affine hull over the EQS part and the interval union over the ITV part.

Bound tightening. In the $\mathcal{CD}$ domain, the bounds of each variable can be obtained from the ITV part of the domain element. The bounds may be changed during domains operations. E.g., when an affine equality is added, the bounds of variables need to be updated. In this paper, we use bound prorogation technique to tighten the bounds.

In fact, each affine equality from the EQS part of the $\mathcal{CD}$ element can be used to tighten the bounds for those variables occurring in the equality. E.g., given an equality $\Sigma_i a_i x_i = b$, if $a_i > 0$, a new candidate lower bound for x_i comes from: $\underline{x}'_i = \lceil (b - \Sigma_{j \neq i} a_j \dot{x}_j)/a_i \rceil$ where $\dot{x}_j = a_j > 0\, ?\, \overline{x}_j : \underline{x}_j$, and a new candidate upper bound for x_i comes from: $\overline{x}'_i = \lfloor (b - \Sigma_{j \neq i} a_j \ddot{x}_j)/a_i \rfloor$ where $\ddot{x}_j = a_j > 0\, ?\, \underline{x}_j : \overline{x}_j$. If

$[\![p := null]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\{p\}\leftarrow 0}$. For each $\mathbf{vec} \in VRVs$ such that $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements: $if(t^{\mathbf{vec}} \geq 1)\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$.
$[\![p := malloc()]\!]^\sharp$	① First, we apply $[\![p := null]\!]^\sharp$. ② Let $\mathbf{vec}' \overset{\text{def}}{=} 0/_{\{p\}\leftarrow 1}$. We build numerical statements: $\{\ t^{\mathbf{vec}'} := 1;\ \}$.
$[\![free(p)]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\mathcal{I}_{\mathbf{vec}_p^0}\leftarrow 0}$. For each $\mathbf{vec} \in \Gamma$ such that $(\mathbf{vec}\ \&\ \mathbf{vec}_p^0) = \mathbf{vec}_p^0$, • if $\mathbf{vec} = \mathbf{vec}_p^0$, we build numerical statements: $if(t^{\mathbf{vec}_p^0} \geq 1)\{\ t^{0...0} := t^{0...0} + t^{\mathbf{vec}_p^0} - 1;\ t^{\mathbf{vec}_p^0} := 0;\ \}$ • otherwise, we build numerical statements: $if(t^{\mathbf{vec}} \geq 1)\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$.
$[\![p := q]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\{p\}\leftarrow q}$. For each $\mathbf{vec} \in \Gamma$ such that $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements: $if(t^{\mathbf{vec}} \geq 1)\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$.
$[\![p := q \rightarrow next]\!]^\sharp$	① First, we apply $[\![p := null]\!]^\sharp$. ② Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\{p\}\leftarrow q}$. For each $\mathbf{vec} \in \Gamma$ such that $\mathbf{vec}_{\{pq\}} = 01$, • if $\mathbf{vec} = \mathbf{vec}_q^0$, we build numerical statements $if(t^{\mathbf{vec}} \geq 1)\{t^{\mathbf{vec}'} := t^{\mathbf{vec}} - 1; t^{\mathbf{vec}} := 1;\}\ \ else\{P' := \top;\}$ • otherwise, we build numerical statements $\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$.
$[\![p \rightarrow next := null]\!]^\sharp$	Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\mathcal{I}_{\mathbf{vec}_p^0}\leftarrow 0}$. For each $\mathbf{vec}$ such that $(\mathbf{vec}\ \&\ \mathbf{vec}_p^0) = \mathbf{vec}_p^0$, • if $\mathbf{vec} = \mathbf{vec}_p^0$, we build numerical statements: $if(t^{\mathbf{vec}_p^0} \geq 1)\{\ t^{0...0} := t^{0...0} + t^{\mathbf{vec}_p^0} - 1;\ t^{\mathbf{vec}_p^0} := 1;\ \}\ else\{P' := \top;\}$ • otherwise, we build numerical statements: $if(t^{\mathbf{vec}} \geq 1)\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\ \}$.
$[\![p \rightarrow next := q]\!]^\sharp$	① First, we apply $[\![p \rightarrow next := null]\!]^\sharp$. ② Let $\mathbf{vec}' \overset{\text{def}}{=} \mathbf{vec}/_{\mathcal{I}_{\mathbf{vec}_p^0}\leftarrow q}$. For each $\mathbf{vec} \in \Gamma$ such that $\mathbf{vec}[q] = 1$ and $\mathbf{vec}' \neq \mathbf{vec}$, we build numerical statements: $\{\ t^{\mathbf{vec}'} := t^{\mathbf{vec}'} + t^{\mathbf{vec}};\ t^{\mathbf{vec}} := 0;\}$

Figure 3: Abstract assignment transfer function over shapes

the new candidate bounds are tighter, then x_i's bounds are updated. This process can be repeated for each variable in that equality and for each equality in the EQS system.

Widening. The height of the lattice of affine equalities is finite, but intervals do not satisfy the ascending chain condition. Thus, to cope with loops, a widening operator is needed to ensure the convergence of fixpoint computations over the $\mathcal{CD}$ domain. Given two $\mathcal{CD}$ elements P, P' satisfying $P \sqsubseteq P'$,

$$P \nabla P' \overset{\text{def}}{=} EQS(P') \sqcap (ITV(P) \nabla_i ITV(P')).$$

Since $t^{\mathbf{vec}} \geq 0$ always holds, we refine the widening over intervals as:

$$[a,b]\nabla_i[c,d] \overset{\text{def}}{=} \begin{cases} [a, b \geq d?b : +\infty] & \text{if } a \leq c, \\ [1, b \geq d?b : +\infty] & \text{else if } a > c \geq 1, \\ [0, b \geq d?b : +\infty] & \text{otherwise.} \end{cases}$$

Here we take 1 as a special threshold for the interval widening, since pointer variable p may use $p \rightarrow next$ to access the successive node, which may cause null pointer dereference.

Furthermore, we use $EQS(P')$ to tighten the bounds of variables obtained by $ITV(P)\nabla_i ITV(P')$ after widening. To avoid the well-known convergence problem of interaction between reduction and widening[19], we perform bound tightening after widening only for finite times. However, since the height of the lattice of affine equalities is finite and each counter variable is bounded by 0 from below, the non-convergence problem will not be serious in our case.

Example 3. Consider again the program fragment shown in Example 2. Assume q points to a list with length 9 before calling $traverse(q)$ and the variable ordering is $p \prec q$. At program point ③, we get in the first iteration the $\mathcal{CD}$ element $P_1 = \{t^{10} = 1, t^{11} = 8; t^{10} \in [1,1], t^{11} \in [8,8]\}$ and in the second iteration $P_2 = \{t^{10} = 2, t^{11} = 7; t^{10} \in [2,2], t^{11} \in [7,7]\}$. And we have

$$\begin{aligned} P_1 \sqcup P_2 &= \{t^{10} + t^{11} = 9; t^{10} \in [1,2], t^{11} \in [7,8]\}, \\ P_1 \nabla (P_1 \sqcup P_2) &= \{t^{10} + t^{11} = 9; t^{10} \in [1,+\infty], t^{11} \in [1,8]\} \\ &= \{t^{10} + t^{11} = 9; t^{10} \in [1,8], t^{11} \in [1,8]\}. \end{aligned}$$

4.3 Transfer Function over Shapes

Test transfer function over shapes. In this paper, we consider only four basic kinds of test condition over pointer variables: $p == null, p == q, p \neq null, p \neq q$. Other complex conditions can be obtained by introducing auxiliary pointer variables and composing basic conditions via logical operators. Let P be the input $\mathcal{CD}$ element before and P' be the resulting CD element after applying the transfer function.

1. $[\![p == null]\!]^\sharp$: When $p == null$ holds, it means that pointer variable p does not point to any lists and thus can not reach any list nodes, i.e.,

$$\forall t^{\mathbf{vec}}.VEC(t^{\mathbf{vec}})[\mathcal{I}_p] = 1 \rightarrow t^{\mathbf{vec}} = 0$$

where $\mathcal{I}_p$ denotes the index of the bit corresponding to pointer variable p. In the $\mathcal{CD}$ domain, we add constraints $t^{\mathbf{vec}} = 0$ to P for those $t^{\mathbf{vec}}$ satisfying $VEC(t^{\mathbf{vec}})[\mathcal{I}_p] = 1$. Then we check the emptiness of P', and tighten variable bounds.

2. $[\![p == q]\!]^\sharp$: When $p == q$ holds, it means that pointer variables p, q are alias. Hence,

$$\forall t^{\mathbf{vec}}.VEC(t^{\mathbf{vec}})[\mathcal{I}_p] \oplus VEC(t^{\mathbf{vec}})[\mathcal{I}_q] = 1 \rightarrow t^{\mathbf{vec}} = 0$$

where $\oplus$ denotes the bitwise XOR operation. In the $\mathcal{CD}$ domain, we add constraints $t^{\mathbf{vec}} = 0$ to P for those $t^{\mathbf{vec}}$ satisfying $VEC(t^{\mathbf{vec}})[\mathcal{I}_p] \oplus VEC(t^{\mathbf{vec}})[\mathcal{I}_q] = 1$. Then we check the emptiness of P' and tighten variable bounds.

3. $[\![p \neq null]\!]^\sharp$: When $p \neq null$ holds, it means that pointer variable p does point to some list node. Hence,

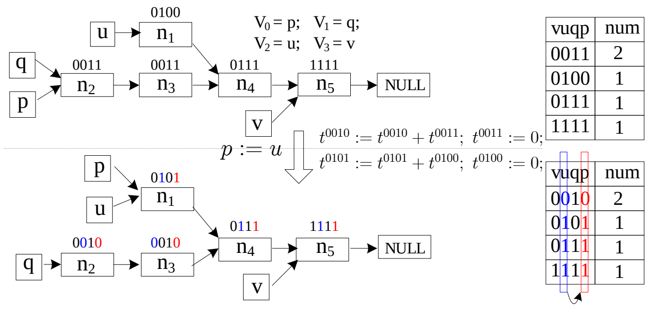
$$\sum_{\mathbf{vec}[\mathcal{I}_p]=1} t^{\mathbf{vec}} \geq 1$$

In the $\mathcal{CD}$ domain, if $t^{\mathbf{vec}} = 0$ holds for all $\mathbf{vec}$ satisfying $\mathbf{vec}[\mathcal{I}_p] = 1$, we put $P' = \bot$. Otherwise, we use the constraint $\sum_{\mathbf{vec}[\mathcal{I}_p]=1} t^{\mathbf{vec}} \geq 1$ to tighten variable bounds.

4. $[\![p \neq q]\!]^\sharp$: When $p \neq q$ holds, it means that there exist at least one list nodes that p and q do not point to at the same time. Hence,
$$\sum_{\mathbf{vec}[\mathcal{I}_p] \oplus \mathbf{vec}[\mathcal{I}_q]=1} t^{\mathbf{vec}} \geq 1$$
In the $\mathcal{CD}$ domain, if $t^{\mathbf{vec}} = 0$ holds for all **vec** satisfying $\mathbf{vec}[\mathcal{I}_p] \oplus \mathbf{vec}[\mathcal{I}_q] = 1$,we put $P' = \bot$. Otherwise, we use $\sum_{\mathbf{vec}[\mathcal{I}_p] \oplus \mathbf{vec}[\mathcal{I}_q]=1} t^{\mathbf{vec}} \geq 1$ to tighten the variable bounds.

Assignment transfer function over shapes. We consider the assignment transfer functions in the form of $P' = [\![astmt]\!]^\sharp(P)$, where *astmt* denotes an assignment statement of shapes. Let $\mathbf{vec}/_{\mathcal{I}\leftarrow 0}$ denote the bitwise substitution of those bits in $\mathcal{I}$ with value 0, $\mathbf{vec}/_{\mathcal{I}\leftarrow q}$ denote the bitwise substitution of those bits in $\mathcal{I}$ with the value of the corresponding bit of variable q, and $\mathbf{vec}_{pq}$ denote the projection of **vec** on positions of p and q. The abstract semantics of assignment transfer function over shapes is shown in Fig. 3. The main idea here is to transform an assignment over shapes into a series of numerical statements over counter variables, according to the changing of the shape.

Example 4. Consider the assignment transfer function $[\![p := u]\!]^\sharp$ over the list shown in Example 1. As depicted in Fig. 4, before applying $[\![p := u]\!]^\sharp$, we have $P = \{t^{0011} = 2, t^{0100} = 1, t^{0111} = 1, t^{1111} = 1; t^{0011} \in [2, 2], t^{0100} \in [1, 1], t^{0111} \in [1, 1], t^{1111} \in [1, 1]\}$. According to the semantics of $[\![p := u]\!]^\sharp$, we know that the bit vector 0011 changes to 0010, and thus we construct the following numeric assignments: $if(t^{0011} \geq 1)\{\ t^{0010} := t^{0010} + t^{0011};\ t^{0011} := 0;\ \}$. Similarly, for the changement from 0100 to 0101, we build numerical assignments: $if(t^{0100} \geq 1)\{\ t^{0101} := t^{0101} + t^{0100};\ t^{0100} := 0;\ \}$. Finally, after performing all the above assignment transfer functions over counters, we will get $P' = \{t^{0010} = 2, t^{0101} = 1, t^{0111} = 1, t^{1111} = 1; t^{0010} \in [2, 2], t^{0101} \in [1, 1], t^{0111} \in [1, 1], t^{1111} \in [1, 1]\}$.


u
n1
0100
q
p
n2
0011
n3
0011
n4
0111
n5
1111
NULL
v
V0 = p; V1 = q;
V2 = u; V3 = v
p := u
$t^{0010} := t^{0010} + t^{0011};\ t^{0011} := 0;$
$t^{0101} := t^{0101} + t^{0100};\ t^{0100} := 0;$
0101
0010
0010
0111
1111

vuqp	num
0011	2
0100	1
0111	1
1111	1

vuqp	num
0010	2
0101	1
0111	1
1111	1

Figure 4: Example of assignment transfer function

Maintaining $\mathbf{vec}^0_p$. From above, we see that the information of $\mathbf{vec}^0_p$ is important for transforming shape assignments that involve "*next*" field to numerical assignments. Recall that $\mathbf{vec}^0_p$ specifies the bit vector that the pointer variable p directly points to. In the concrete semantics, $\mathbf{vec}^0_p$ can be computed from environment of auxiliary counter variables, i.e., the least bit vector **vec** such that $\mathbf{vec}[p] = 1 \wedge t^{\mathbf{vec}} > 0$. However, in the abstract semantics, due to precision loss, we may not have enough information to determine such a vector and may only know some **vec** satisfying $\mathbf{vec}[p] = 1 \wedge t^{\mathbf{vec}} \geq 0$. In this case, there may be 2 subcases:

- if $\mathbf{vec}[p] = 1 \wedge t^{\mathbf{vec}} > 0$, then $\mathbf{vec}^0_p = \mathbf{vec}$,
- else if $\mathbf{vec}[p] = 1 \wedge t^{\mathbf{vec}} = 0$, then $\mathbf{vec}^0_p$ is some other vector $\mathbf{vec}'$ greater than **vec**.

For the sake of precision, we maintain a set of possible $\mathbf{vec}^0_p$'s, denoted as Γ^0_p, for each pointer variable p. We redefine the transfer functions based on Γ^0_p. The main idea here is to first apply the transfer function separately on each $\mathbf{vec}^0_p \in \Gamma^0_p$ and then perform join over the results computed from each $\mathbf{vec}^0_p$. We also define rules to update Γ^0_p after each transfer function.

4.4 Extension Discussions

Abstraction over list contents. The $\mathcal{CD}$ domain can be easily extended to support abstractions over list contents. Following the same idea of introducing counter variables, for each $\mathbf{vec} \in \Gamma$, we could introduce an auxiliary content variable $d^{\mathbf{vec}}$ to abstract the contents of the list nodes whose VRV is **vec**. If the list contents are of numerical data type, we could apply numerical abstractions to $d^{\mathbf{vec}}$, similarly to what we do over $t^{\mathbf{vec}}$. However, to be sound, we can only apply weak update semantics to handle assignments to $p \rightarrow data$. In order to obtain more precise information over $p \rightarrow data$, we need to extend our bit-vector shape abstraction to distinguish the first element from the rest elements in a list segment.

Extension to circular lists and doubly linked lists. Until now, we have considered only singly linked lists without cycle. To deal with circular lists, we first introduce a specific tag bit for each VRV to denote whether the list nodes abstracted by this VRV are involved within a cycle. If there is a cycle, we employ a so-called "cut" operation to choose a cutpoint to cut the cycle at some next edge, which transforms a circular list into a non-circular one. To deal with doubly linked lists that are well-founded [14], we maintain reachability properties for the *next* field and the *prev* field separately. In other words, we maintain different VRV sets for *next* and *prev*. And we do communication and propagation between the two fields when needed.

5. EXPERIMENTS

We have developed a prototype tool for analyzing list manipulating programs based on the Apron [12] numerical abstract domain library and the Interproc [15] static analyzer. We implemented our $\mathcal{CD}$ domain inside Apron. Since Interproc uses the Spl input language which supports only numeric (integer or real) variables, inspired by CINV[3], we encode our programs on lists via Spl. Pointer variables of list type are coded by real variables while data variables are encoded by integer variables. The constant NULL is encoded by value 0.0. And the operations on pointers are encoded using operations on real variables. E.g., $p := q \rightarrow next$ is encoded by $p = cast_{f,n}(q)$, and $p \rightarrow next := q$ is encoded by $p = cast_{d,n}(q)$.

To exemplify the ability of invariant synthesis of our $\mathcal{CD}$ domain, let us consider an example program copy_and_delete1 (which for sake of space allows statements like $p := p \rightarrow next$ and is a simplified version of copy_and_delete2 in Fig. 5) together with the generated invariants by the $\mathcal{CD}$ domain, as shown in Fig. 6. The program first copies one list to another and then deletes both simultaneously. Suppose the initial length of the input list *xList* is 9 and the variable ordering is $pList \prec qList \prec xList \prec yList$. From the invariants after line 7, we can see: (1) Pointer variables *pList* , *qList* are alias while *xList* , *yList* are alias; (2) The two list respectively pointed to by *pList* and *xList* are of the same length; (3) The bounds of counter variables are strictly positive, which indicates that the operations on lines 8-10 are free of null pointer dereference. Finally, the special auxiliary variable $t^{0\cdots 0}$ equals to 0 at all the program points, which proves the absence of memory leak in the program.

Our experiments were conducted on a selection of benchmark examples, some of which are taken from [3]. These benchmark ex-

Program		$\mathcal{CD}$ domain		Key loop invariants
name	#pvars	#iterat.	$time(ms)$	
create	2	4	2	$t^{10} = n$
traverse	3	5	3	$t^{010} + t^{011} = n$
reverse	3	5	3	$t^{001} + t^{010} = n$
length_equal	5	5	8	$t^{00001} + t^{00011} = n;$ $t^{00001} - t^{00100} = 0; t^{00011} - t^{01100} = 0$
merge	5	5	32	$t^{00001} + t^{10001} = n; t^{01000} + t^{10001} = n$
copy_and_delete2	5	9	76	$t^{00011} + t^{11000} = n; t^{01000} + t^{11000} = n$
dispatch	6	5	227	$t^{000010} + t^{000100} + t^{001000} = n$

Figure 5: Experimental results for benchmark examples

```
void copy_and_delete(List* $xList$) {
/* assume \length($xList$)==9; */
  List* $yList, pList, qList$;
$/ * t^{0100} = 9; t^{0100} \in [9, 9]; \heartsuit * /$
  $yList = xList$;    $qList = pList = null$;
$/ * t^{0100} + t^{1100} = 9, t^{0011} + t^{1100} = 9; t^{0100} \in [1, 9], t^{1100} \in [0, 9],$
    $t^{0011} \in [0, 9]; \heartsuit * /$
  while ($yList$ != $null$){
$/ * t^{0100} + t^{1100} = 9, t^{0011} + t^{1100} = 9; t^{0100} \in [1, 8], t^{1100} \in [1, 9],$
    $t^{0011} \in [0, 8]; \heartsuit * /$
      $yList = yList \rightarrow next$; $qList$ = malloc();
      $qList \rightarrow next = pList$; $pList = qList$;}
$/ * t^{0011} = 9, t^{0100} = 9; t^{0011} \in [9, 9], t^{0100} \in [9, 9]; \heartsuit * /$
  $yList = xList$;
$/ * t^{0011} - t^{1100} = 0; t^{0011} \in [0, 9], t^{1100} \in [0, 9]; \heartsuit * /$
  while ($yList$ != $null$){
$/ * t^{0011} - t^{1100} = 0; t^{0011} \in [1, 9], t^{1100} \in [1, 9]; \heartsuit * /$
      $yList = yList \rightarrow next$; $qList = qList \rightarrow next$;
      free($xList$); free($pList$);
      $xList = yList$; $pList = qList$;
  }
$/ * \forall \mathbf{vec}.t^{\mathbf{vec}} = 0 * /$
}
```

Figure 6: Example program *copy_and_delete*1 **and the generated invariants. The notation ♡ means $t^{vec} = 0$ for any t^{vec} that does not appear in the annotated invariant.**

amples contain commonly used operations over lists, such as create, traverse, reverse, merge, copy, delete and dispatch. Experimental results are shown in Fig. 5. For each program, "#pvars" indicates the total number of pointer variables in the program. "#iterat." gives the number of increasing iterations during the analysis.

Invariants. These benchmark programs involve relational properties among lengths of list segments. Our $\mathcal{CD}$ domain that is based on intervals and affines equalities, is able to find interesting affine equality relations and bounds of lengths of list segments. In Fig. 5, the column "Key loop invariants" gives some important affine equality relations among list segments found by our $\mathcal{CD}$ domain inside loops.

Performance. The column "time(ms)" in Fig. 5 presents the analysis times in milliseconds when the analyzer is run on a 3.1GHz PC with 4GB of RAM running Fedora 12. The results show that the analysis upon $\mathcal{CD}$ seems efficient, but increasing the number of pointer variables degrades the computation cost a lot. Currently, we use the dense representation in Apron to represent the matrices in the affine equality domain. During our experiments, we found that matrices are in fact rather sparse, almost linear to the number of pointer variables in most cases. Our future implementation will consider using the sparse representation.

6. RELATED WORK

Shape Abstractions. Programs manipulating lists have gained much attention within the past decade [9] [18] [4]. And various abstractions have been used for analyzing shape properties of lists and dynamically linked data structures that are more general, such as canonical abstraction [23], boolean heaps [21], separation logic [8], etc. One work that is close to our shape abstraction is boolean heaps [21] by Podelski et al. It adopts also the concept "bit-vector" and utilizes sets of bit-vectors to encode boolean heaps. Their approach is general for modeling all kinds of relations in heaps by using proper heap predicates, while in this paper we focus on lists and utilize one basic predicate (*Reach*). In addition, our approach maintains automatically numeric information (length of list). Recently, Gulwani et al. [11] propose an abstract domain that allows representation of must and may equalities among pointer expressions. Our work uses also equalities but to track the numerical properties over the number of list nodes. Hguyen et al. [20] propose an approach based on separation logic that can precisely track shape and size properties. Their approach is general for a wide range of data structures (e.g., trees, priority heap, lists, etc.) but needs user-defined shape predicates and does not infer invariants.

Combining shape and numerical abstractions. Recently, much attention has been focused on combining shape and numerical abstractions [4][5][10][22]. Bouajjani et al. [2][3][4] utilize counter automata as an abstract model for lists, and propose a framework for combining a heap abstraction with various abstractions over the sequences of data in a list. Their method maintains the exact data stored in a list segment as well as their sequences and thus can discover relational properties over list contents. Compared with the counter automata model, our heap abstraction based on bit-vectors is quite lightweight. Moreover, our approach can also find alias among pointer variables and detect potential null pointer dereference as well as memory leak. More recently, Gulwani et al. [10] propose a general combination framework for tracking relationships between sizes of memory partitions. It combines a set domain (for tracking memory partitions) with a set cardinality domain (for tracking relations between cardinalities of the partitions) via reduced products. Our work encodes the shape abstraction by bit-vectors that itself can be considered as numerical values, which makes it easy to build a combined domain based on numerical abstractions taking into account the semantics of shape abstractions, without resort to reduced product.

Reducing heap-manipulating to numerical programs. Magill et al. [16][17] propose a method to automatically transform heap manipulating programs into numeric ones while keeping the desired property. Separating shape abstraction and numerical abstraction has clear engineering advantages to make use of existing numeric reasoning tools. However, the transformation is unidirectional and thus may lose precision especially when the shape aspect and the numerical aspect interact in complicated ways. Our work allows bidirectional communication between shape and numerical aspects.

7. CONCLUSION

We have presented an approach in the framework of abstract interpretation for analyzing list-manipulating programs. The main idea is to combine heap and numerical abstractions. The structural information of the shape of a list is encoded in a lightweight way via bit vectors, one for each list segment, while numerical relations among the number of list nodes in these segments are tracked by numerical abstract domains. We have instantiated our approach by establishing a combination domain of intervals and affine equalities to infer relations over the length of list segments. A key benefit of our approach is the ability to leverage the power of the state-of-the-art numerical abstract domains to analyze intricate properties of list-manipulating programs.

Future work will consider extending our approach to infer properties over the content of lists, e.g., sortedness, no duplicated elements. Inferring non-trivial relational properties over list contents requires reasoning over inter-segment relations among different segments and intra-segment relations among elements in the same segment.

8. REFERENCES

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, volume 4590 of *LNCS*. Springer, 2007.

[2] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. *Formal Methods in System Design*, 38(2):158–192, 2011.

[3] A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *CAV'10*, volume 6174 of *LNCS*, pages 72–88. Springer, 2010.

[4] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, volume 7148 of *LNCS*, pages 1–22. Springer, 2012.

[5] B. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL'08*, pages 247–260. ACM, 2008.

[6] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the 2nd International Symposium on Programming*, pages 106–130. Dunod, Paris, 1976.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252. ACM Press, 1977.

[8] D. Distefano, P.W.O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3902 of *LNCS*, pages 287–302. Springer, 2006.

[9] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS*, volume 1824 of *LNCS*, pages 115–134. Springer, 2000.

[10] S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL'09*, pages 239–251. ACM, 2009.

[11] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, volume 4590 of *LNCS*, pages 379–392. Springer, 2007.

[12] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV'09*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[13] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[14] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *ACM POPL'06*, pages 115–126. ACM Press, 2006.

[15] G. Lalire, M. Argoud, and B. Jeannet. Interproc. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/.

[16] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS'07*, volume 4634 of *LNCS*, pages 419–436. Springer, 2007.

[17] S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL'10*, pages 211–222. ACM, 2010.

[18] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, volume 3385 of *LNCS*, pages 181–198. Springer, 2005.

[19] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[20] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, volume 4349 of *LNCS*, pages 251–266. Springer, 2007.

[21] A. Podelski and T. Wies. Boolean heaps. In *SAS*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.

[22] S. Qin, G. He, C. Luo, and W. Chin. Loop invariant synthesis in a combined domain. In *ICFEM'10*, volume 6447 of *LNCS*, pages 468–484. Springer, 2010.

[23] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.